Non-Blocking Interpolation Search Trees with Doubly-Logarithmic Running Time

Trevor Brown
University of Waterloo
Canada
me@tbrown.pro

Aleksandar Prokopec
Oracle Labs
Switzerland
aleksandar.prokopec@gmail.com

Dan Alistarh
Institute of Science and Technology
Austria
dan.alistarh@ist.ac.at

Abstract

Balanced search trees typically use key comparisons to guide their operations, and achieve logarithmic running time. By relying on numerical properties of the keys, *interpolation search* achieves lower search complexity and better performance. Although interpolation-based data structures were investigated in the past, their non-blocking concurrent variants have received very little attention so far.

In this paper, we propose the first non-blocking implementation of the classic interpolation search tree (IST) data structure. For arbitrary key distributions, the data structure ensures worst-case $O(\log n + p)$ amortized time for search, insertion and deletion traversals. When the input key distributions are *smooth*, lookups run in expected $O(\log \log n + p)$ time, and insertion and deletion run in expected amortized $O(\log \log n + p)$ time, where p is a bound on the number of threads. To improve the scalability of concurrent insertion and deletion, we propose a novel parallel rebuilding technique, which should be of independent interest.

We evaluate whether the theoretical improvements translate to practice by implementing the concurrent interpolation search tree, and benchmarking it on uniform and non-uniform key distributions, for dataset sizes in the millions to billions of keys. Relative to the state-of-the-art concurrent data structures, the concurrent interpolation search tree achieves performance improvements of up to 15% under high update rates, and of up to 50% under moderate update rates. Further, ISTs exhibit up to 2× less cache-misses, and consume $1.2 - 2.6\times$ less memory compared to the next best alternative on typical dataset sizes. We find that the results are surprisingly robust to distributional skew, which suggests that our data structure can be a promising alternative to classic concurrent search structures.

CCS Concepts • **Theory of computation** → **Concurrent algorithms**; *Shared memory algorithms*; • **Computing methodologies** → **Concurrent algorithms**;

Keywords concurrent data structures, search trees, interpolation, non-blocking algorithms

1 Introduction

Efficient search data structures are critical in practical settings such as databases, where the large amounts of underlying data are usually paired with high search volumes, and with high amounts of concurrency on the hardware side, via tens or even hundreds of parallel threads. Consequently, there has been a significant amount of research on efficient *concurrent* implementations of search data structures.

For search data structures supporting predecessor queries, which are the focus of this work, such as binary search trees (BSTs) or balanced search trees, efficient implementations have been well researched and are relatively well understood, e.g. [9, 13, 22, 36]. However, these classic search data structures are subject to the fundamental *logarithmic* complexity thresholds (in the number of keys n), even in the average case, which limits their performance for large key sets, in the order of millions or even billions of keys. In the sequential case, elegant and non-trivial techniques have been proposed to reduce average-case complexity, by leveraging properties of the key space, or of the key distribution. With one notable exception [37], these techniques are significantly less well understood for concurrent implementations.

This paper revisits this area, and provides the first efficient, non-blocking concurrent implementation of an *interpolation search tree* data structure [34], called the C-IST. The C-IST is *dynamic*, in that it supports concurrent searches, insertions and deletions. Interpolation search trees, presented in the next section, have amortized worst-case $O(\log n)$ time for standard operations, but achieve $O(\log \log n)$ expected amortized time complexity for insert and delete, and $O(\log \log n)$ expected time for search, by leveraging smoothness properties of the key distribution [34]. Our concurrent implementation preserves these properties with high probability.

PPoPP '20, February 22–26, 2020, San Diego, CA, USA

ACM ISBN 978-1-4503-6818-6/20/02…$15.00
https://doi.org/10.1145/3332466.3374542

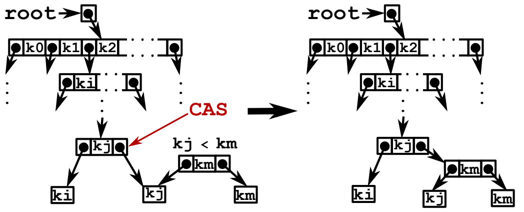
To ensure correctness, non-blocking progress, and scalability in the concurrent setting, we introduce several new techniques relative to sequential ISTs. Specifically, our contributions are as follows:

- We describe the first non-blocking concurrent interpolation search tree (C-IST) based on atomic compare-and-swap (CAS) instructions (Section 2), with expected lookup time $O(\log \log n + p)$, and expected amortized $O(\log \log n + p)$ time for insert and delete.
- We design a *parallel*, *non-blocking* rebuilding algorithm to provide fast and scalable periodic rebuilding for C-ISTs (Section 3). We believe that this technique is applicable to other concurrent data structures that require rebuilding.
- We prove the correctness, non-blocking and complexity properties of the C-IST (Section 4).
- We provide a C-IST implementation in C++, and compare its performance against concurrent (a, b)-trees [13], Natarajan and Mittal's concurrent BSTs [36], and Bronson's concurrent AVL trees [10] (Section 5). We report performance improvements of 15% − 50% compared to (a, b)-trees (the prior best-performing concurrent search tree) on large datasets, and improvements of up to 3.5× compared to the other concurrent trees, depending on the proportion of updates. We also analyze the average depth and cache-miss behavior, present a breakdown of the execution time, show the impact of the parallel rebuilding algorithm, and compare memory footprints.

2 Concurrent Interpolation Search Tree

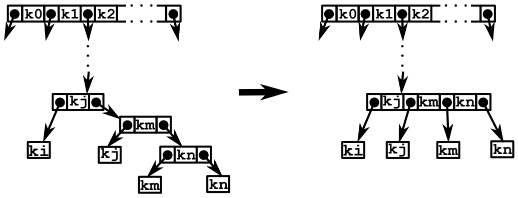
2.1 Examples and Overview

We illustrate how concurrent interpolation search trees work using several examples. Examine the first tree in the following figure. Each inner node consists of a set of d pointers to child nodes, and $d - 1$ keys that are used to drive the search. We say that the node's *degree* is d. The top node usually has the highest degree, and the degree of a node decreases as it gets deeper in the tree (explained precisely below). The tree is *external*, meaning that the keys are stored in the leaf nodes. The illustration shows a subset of nodes – the missing nodes are represented with $\cdots$ symbols.



Consider the task of inserting a key k_m, such that $k_j < k_m < k_l$, where k_j and k_l are existing keys in the tree. The figure shows a tree in which k_j is contained in a leaf node on the bottom. Insertion finds the leaf corresponding to k_j, such that k_m is the successor of k_j, and then allocates a new inner node that holds both k_j and k_m. Finally, the old pointer in the parent is atomically changed with a `CAS` instruction to point to the new node.

Without rebalancing, the tree can become arbitrarily deep. Therefore, insertion must periodically rebalance parts of the tree. The following figure shows the tree after inserting an additional key k_n, such that $k_i < k_j < k_m < k_n$. The subtree at the bottom, which contains the keys k_i, k_j, k_m and k_n, is sufficiently imbalanced, and it should be replaced with a more balanced tree. Rebalancing creates a new subtree that contains the same set of keys. After rebalancing, the subtree consists of a single inner node of degree 4, as shown on the right. Note that deletions also periodically rebalance the subtrees.



There are several challenges with this making this approach concurrent. First, concurrent modifications and rebalancing must correctly synchronize so that all operations remain non-blocking, while searches remain wait-free. Second, the rebalancing of any subtree must not compromise the scalability of the other operations. Finally, concurrent rebalancing must, when the probability distribution of the input keys is smooth [34], ensure that the operations run in amortized $O(\log \log n)$ time.

2.2 Data Types

The concurrent interpolation search tree consists of the data types shown in Figure 1. The `IST` data type represents the interpolation search tree with the single member `root`, which points to the root node. Initially, the root node points to an empty leaf node, whose type is `Empty`. The `Single` data type represents a leaf node with a single key and an associated value, and the `Inner` data type represents inner nodes, as illustrated on the right of Figure 1.

In addition to holding the search keys, and the pointers to the child nodes, the `Inner` data type contains the node's `degree`, and a field called `initSize`, which contains the number of keys that were in the corresponding subtree when this node was created. Apart from the child pointers, these fields are set on creation, and not subsequently modified.

`Inner` also contains two volatile fields, `count` and `status`, which are used to coordinate rebuilding. The `count` field holds the number of updates that were performed in the subtree rooted at this node since it was created. The `status` field consists of an integer and two booleans – it is initially zero, and then changes to a non-zero value to indicate that this node will be replaced during a rebuilding operation.

```
struct IST is
  root: Node

struct Single: Node is
  key: KeyType
  val: ValType

struct Inner: Node is
  initSize: int
  degree: int
  keys: KeyType[]
  children: Node[]
  status: [int, bool, bool]
  count: int

struct Empty: Node is

struct Rebuild: Node is
  target: Inner
  parent: Inner
  index: int
```

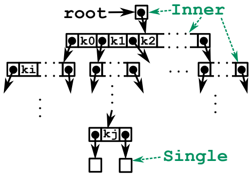


Figure 1. Data Types

The Rebuild data type contains information about a subtree-rebuilding operation. It contains a pointer called target to the root of the subtree to rebuild, a pointer to its parent, and the index of the target in the parent node's array of child pointers. The status field and the Rebuild type are further explained in Section 2.4.

To perform correctly, IST operations must maintain certain invariants – informally, these invariants state that there must be a unique, acyclic path to any key, and that the nodes cover disjoint key intervals. They are formally defined below.

Invariant 1 (Key presence). *For any key* $\mathtt{k}$ *reachable in the IST* $\mathtt{I}$*, there exists exactly one path of the form* $\mathtt{I} \xrightarrow{root} \mathtt{n}_0 \xrightarrow{children[i_0]} (\mathtt{n}_1 \mid \mathtt{r}_1 \xrightarrow{target} \mathtt{n}_1) \xrightarrow{children[i_1]} \dots \xrightarrow{children[i_{m-1}]} (\mathtt{n}_m \mid \mathtt{r}_m \xrightarrow{target} \mathtt{n}_m)$*, where* $\mathtt{n}_m$ *holds the key* $\mathtt{k}$*,* $\mathtt{r}_m$ *is a* Rebuild *node, and* | *is a choice between two patterns.*

Definition 2.1 (Cover). A root node n *covers* the interval $\langle -\infty, \infty \rangle$. Given an inner node n of degree d that covers the interval $[a, b\rangle$, and holds the keys $k_0, k_1, \dots, k_{d-2}$ in its keys array, its child n.children[i] covers the interval $[k_{i-1}, k_i\rangle$, where we define $k_{-1} = a$ and $k_{d-1} = b$.

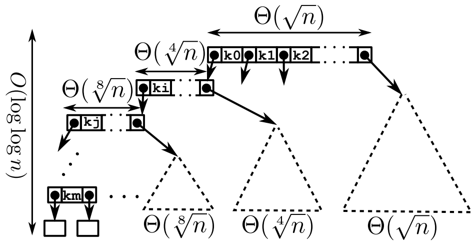
Definition 2.2 (Content). A node n *contains* a key k if and only if the path from the root of the IST I to the leaf with the key k contains the node n. An IST I *contains* a key k if and only if the root contains the key k.

Invariant 2 (Search tree). *If a node* n *covers* $[a, b\rangle$ *and contains a key* k*, then* $\mathtt{k} \in [a, b\rangle$*.*

Invariant 3 (Acyclicity). *There are no cycles in the interpolation search tree.*

Definition 2.3 (Has-key). Relation hasKey(I, k) holds if and only if I satisfies the invariants, and contains the key k.

In the interpolation search tree, the degree d of a node with cardinality n is $\Theta(\sqrt{n})$. In an *ideal IST*, the degree of a node with cardinality n is either $\lfloor \sqrt{n} \rfloor$ or $\lceil \sqrt{n} \rceil$, and the number of keys in each of the node's subtrees is $\Theta(\sqrt{n})$, more specifically, either $\lfloor \sqrt{n} \rfloor$ or $\lceil \sqrt{n} \rceil$. This ensures the $O(\log \log n)$ depth bound. An example of an ideal IST is shown below – the root has degree $\Theta(\sqrt{n})$, its children have degree $\Theta(\sqrt[4]{n})$, its grandchildren have degree $\Theta(\sqrt[8]{n})$ and so on. The interpolation search tree will generally not be ideal after a sequence of insertion and deletion operations, but its subtrees are ideal ISTs immediately after they get rebuilt.



2.3 Insertion and Deletion

As illustrated in Section 2.1, an insertion searches the tree for a Single or Empty node, and then replaces this node with one or two new nodes. An Empty node is replaced with a new Single node that contains the new key, and a Single node is replaced with an inner node.

To track the amount of imbalance in each subtree, the standard IST increments the count for all the inner nodes that lead to that leaf, whenever a key is inserted or deleted at that leaf [34]. Once some count reaches a threshold, the corresponding subtree is rebuilt. Our C-IST implementation avoids contention at the root by using a scalable, quiescently-consistent multicounter [2] at the root.

Once rebalancing is triggered, subsequent insertions and deletions in the corresponding subtree must fail, and help complete the rebalancing before retrying. To ensure this, the rebalancing sets the status field of all the nodes of the target subtree. An insertion atomically checks the status field of an inner node while replacing a child of the inner node. We accomplish this with an atomic double-compare-single-swap (DCSS) primitive, which takes two addresses, two corresponding expected values, and one new value as arguments, and behaves like a CAS that succeeds only if the second address also matches its expected value. DCSS also provides the wait-free DCSS_READ primitive, which can read the fields that can be concurrently modified by a DCSS. Both are efficiently implemented using single-word CASes [3, 28].

Insertion. Figure 2 shows the pseudocode for insert, which traverses the C-IST starting at the root. An interpolation search [38] is done at each node to determine the index of the next child pointer for the given key. This search uses the linear interpolation between the node's minimum and maximum keys to estimate the index (in the node's array of keys) to which the specified key belongs, and does a linear search thereafter. Since the keys array does not change after the creation of an Inner node, interpolationSearch has a sequential implementation, not shown here.

```
1  procedure insert(ist, key, val)
2    path = []                       // Stack that saves the path.
3    n = ist.root
4    while true
5      index = interpolationSearch(key, n)
6      child = DCSS_READ(n.children[index])
7      if child is Inner
8        n = child
9        path.push( [child, index] )
10     else if child is Empty | Single
11       r = createFrom(child, key, val)
12       result =
13         DCSS(n.children[index], child, r, n.status, [0,⊥,⊥])
14       if result == FAILED_MAIN_ADDRESS
15         continue                    // Retry from the same n.
16       else if result == FAILED_AUX_ADDRESS
17         return insert(ist, key, val)  // Retry from the root.
18       else
19         for each [n, index] in path
20           FETCH_AND_ADD(n.count, 1)
21         parent = ist.root
22         for each [n, index] in path
23           count = READ(n.count)
24           if count >= REBUILD_THRESHOLD ∗ n.initSize
25             rebuild(n, parent, index)
26             break                          // exit for
27           parent = n
28         return true
29     else if child is Rebuild
30       helpRebuild(child)
31       return insert(ist, key, val)    // Retry from the root.
```

Figure 2. Insert Operation

Next, insert checks the type of the child node. If child is an inner node, then insert continues the traversal, and at the same time adds the child to the list called path. This list is used to update the counts, as explained shortly. If child is an Empty or a Single node, then insert replaces it with a new node r with one or two keys, respectively, allocated in the createFrom subroutine. The DCSS in line 13 inserts the new node by changing n.children[index] from child to r only if n.status == [0,⊥,⊥].

The DCSS in line 13 fails when n.status ≠ [0,⊥,⊥], and returns the FAILED_AUX_ADDRESS value, which indicates that the node is a part of an ongoing a rebuild. In this case, insert restarts from the root to find the Rebuild node, and help complete the rebuild. The DCSS could also fail if n.children[index] ≠ child, indicating that another insert or delete or rebuilding operation modified the same location. In this case, insert restarts from the same n. If the DCSS is successful, then insert increments the count fields with the FETCH_AND_ADD in line 20.

Finally, the insert searches the ancestors in path for the highest node whose count reached the threshold. The threshold is checked in line 24, where REBUILD_THRESHOLD is set to 0.25 (explained in Section 4 of the corresponding tech report [52]). If such a node exists, then insert calls rebuild to recreate the respective subtree. As explained in Section 2.4, rebuild inserts a Rebuild node into the IST. When other updates see this node, they help complete the rebuild before proceeding.

Deletion. The delete either replaces a Single node with a new Empty node, or does not change the data structure if the key is not present. It is almost identical to insert – the main difference is that when child is an Empty node, delete simply returns false, and when child is a Single, instead of calling createFrom, the node is replaced an Empty if the keys match. The deletion does not shrink Inner nodes – while some Empty nodes can accumulate in the tree, then the rebuilding operations eventually remove them. With our chosen threshold, at most 25% of all nodes can be Empty.

2.4 Partial Rebuilding

When insertion or deletion detects that a subtree rooted at a node *target* (henceforth, the "target subtree") has become sufficiently imbalanced, it rebuilds the subtree, as shown in Figure 3. Rebuilding has four steps. First, a thread announces the intention by creating a Rebuild descriptor, and inserts the descriptor between the *target* and its *parent*. Second, the thread does a preorder traversal of the subtree, and sets a bit in the status field of each node to prevent further updates. Third, the thread creates an ideal IST (rooted at *ideal*) using the old subtree's keys (rooted at *target*). Finally, the old subtree is replaced with the new subtree in the parent.

```
32 procedure rebuild(node, p, i)
33   op = new Rebuild(node, p, i)
34   result = DCSS(p.children[i], node, op, p.status, [0,⊥,⊥])
35   if result == SUCCESS then helpRebuild(op)
36
37 procedure helpRebuild(op)
38   keyCount = markAndCount(op.target)
39   ideal = createIdeal(op.target, keyCount)
40   p = op.parent
41   DCSS(p.children[op.index], op, ideal, p.status, [0,⊥,⊥])
42
43 procedure markAndCount(node)
44   if node is Empty then return 0
45   if node is Single then return 1
46   if node is Rebuild then return markAndCount(op.target)
47   // node is Inner
48   CAS(node.status, [0,⊥,⊥], [0,⊥,⊤])
49   keyCount = 0
50   for index in 0 until length(node.children)
51     child = READ(node.children[index])
52     if child is Inner then
53       [count, finished, started] = READ(child.status)
54       if finished then keyCount += count
55       else keyCount += markAndCount(child)
56     else if child is Single then keyCount += 1
57   CAS(node.status, [0,⊥,⊤], [keyCount,⊤,⊤])
58   return keyCount
```

Figure 3. Rebuild Operation

Implementation. In the first step, the rebuild procedure creates the Rebuild descriptor object, and announces it with the DCSS in line 34. If this DCSS is not successful, then either there is another rebuild in some ancestor, or another

```
struct Rebuild: Node is       struct Inner: Node is
  target: Inner                 initSize: int
  newTarget: Inner              degree: int
  parent: Inner                 keys: KeyType[]
  index: int                    children: Node[]
                                status: [int, bool, bool]
                                count: int
                                nextMark: int
```

Figure 4. Modified Data Types for Collaborative Rebuilding

thread concurrently started the rebuild at the same node – in both cases, the current thread can abort the rebuild.

If the announcement is successful, rebuilding continues in the `helpRebuild` subroutine. If other threads need to update the respective subtree, then they observe the `Rebuild` node, and help by also calling `helpRebuild`. The call to `markAndCount` traverses the subtree, and sets the `status` field of each inner node to $[0, \bot, \top]$ with the `CAS` in line 48. At the same time, `markAndCount` counts the number of keys below each node, and, once all the children are traversed, it stores the total key count in the higher bits of the `status` field in line 57. The count allows computing the node degrees in the new subtree. Note that, since the threads compete to set the same value in `status`, the success of these two `CAS`es does not need to be checked – exactly one thread succeeds.

The `createIdeal` creates an ideal IST, as defined in Section 2.2, and ensures that the degree of each node is $\approx \sqrt{N}$, where N is the number of keys in that node's subtree. We do not show its pseudocode, since it involves no concurrency. We refer the reader to e.g. [34] for the exact description of this procedure.

In the last step, the old subtree is replaced with the new one by the `DCSS` in line 41. If this `DCSS` fails, then either another thread finished the rebuild, or another rebuild started in some ancestor, so no further action is necessary.

3 Concurrent Interpolation Search Tree with Collaborative Rebuilding

The basic rebuilding procedure, described in Section 2.3, suffers from a scalability bottleneck when a lot of threads concurrently modify the IST. Since multiple threads compete to mark the old subtree in the `markAndCount` procedure, and multiple threads create the same new subtree in the `createIdeal` procedure from Figure 7, part of the work can be duplicated due to contention. To address this, we designed and implemented a collaborative rebuilding algorithm, in which threads mark and rebuild the subtree in parallel.

3.1 Fast Collaborative Rebuilding

To enable threads to perform rebuilding *collaboratively*, we make several changes in the algorithm. First, we replace the `markAndCount` procedure with a new procedure called `markAndCountCollaborative`, in which helpers attempt to process different parts of the data structure in parallel, and carefully avoid duplicating the work.

```
43 procedure markAndCountCollaborative(node)
     // ... same as markAndCount until line 48, but with
     // recursive calls to markAndCountCollaborative ...
49   if node.degree > COLLABORATION_THRESHOLD
50     while true
51       index = FETCH_AND_ADD(node.nextMark, 1)
52       if index >= node.degree then break
53       markAndCountCollaborative(node.children[index])
     // ... same as markAndCount from line 49, but with
     // recursive calls to markAndCountCollaborative ...
```

Figure 5. The `markAndCountCollaborative` Procedure

Second, we replace the call to `createIdeal` inside the procedure `helpRebuild` in Figure 3 with a call to a new procedure `createIdealCollaborative`, in which a new root of the subtree is first created (which initially contains only `null`-pointers) and *announced*. For this purpose, we add the `newTarget` field to the `Rebuild` data type, as shown in Figure 4, to store the root node of the new subtree. Each `null`-pointer in the new root of the subtree represents a "job" that a thread can perform by building the corresponding subtree (and changing the `null`-pointer to point to this new subtree). Of course, many of these jobs can be performed in parallel. This way, until the new ideal IST is complete, the `newTarget` node serves as a sort of lock-free work queue. Finally, we add the `nextMark` field to `Inner` nodes, which is used in collaborative marking.

These subtlety of these changes is to distribute work among threads while preserving lock-freedom, which mandates that all work is done eventually, even if some threads block.

The collaborative rebuilding algorithm is illustrated in Figure 7, which we explain in the following paragraphs.

Collaborative marking algorithm. Similar to the basic algorithm from Section 2, the collaborative rebuilding algorithm starts by setting the `status` field of all the nodes in the subtree that must be rebuilt. The main difference in the collaborative marking algorithm is that it allows the helping threads to mark parts of the subtree in parallel. The `markAndCountCollaborative` procedure, shown in Fig. 5, starts by setting the low boolean of the `status` field, and is the same as the basic `markAndCount` from Fig. 3 until line 48. If the number of children of the node is larger than the `COLLABORATION_THRESHOLD` value (experimentally set to 48), the marking repetitively invokes the atomic `FETCH_AND_ADD` instruction on the `nextMark` field, to get the index of the next free child that can be recursively marked (line 49). This allows multiple threads to concurrently mark the distinct children, which reduces the memory contention.

The rest of the `markAndCountCollaborative` procedure is exactly the same as the `markAndCount` procedure from Fig. 3 after line 49. In particular, after executing the loop in lines 50-53 of Fig. 5, collaborative marking does another pass through the node's `children` array to help the other threads that are slow. In this second pass (lines 50-55 of Fig. 3), the thread recursively marks those children whose key-count was not yet computed. This second pass is necessary to preserve lock-freedom – if any of the other threads halts, the marking will complete in a finite number of steps.

Collaborative marking example. The collaborative marking is illustrated in Fig. 7. The `Rebuild` object is first announced in Fig. 7A. At this point, the `status` fields of all the inner nodes in the `target` subtree are set to `[0,⊥,⊥]` (shown in the rightmost box of each node), indicating that the marking has not started in any of those nodes. In Fig. 7B, thread p executed `FETCH_AND_ADD` and decided to mark the child at index `0`, while threads q and r are marking children at indices `1` and `3`, respectively. Thread q has completed the marking (indicated by the `[2,⊤,⊤]` in the `status` field of the corresponding child), and can now help threads p and r to complete the marking and set the key counts of their children. In Fig. 7C, all the threads have finished marking, and the `target` node has the `status` field set to `[9,⊤,⊤]`. At this point, no more concurrent modifications of the `target` subtree are possible, and `target` can be traversed without synchronization for the purposes of creating a new subtree.

Collaborative building. After the `target` subtree is marked, and the total key-count is known, the algorithm allocates the root node of the new subtree. Once again, if the key-count is below the `COLLABORATION_THRESHOLD`, the entire subtree is created without collaboration with the call to the `createIdeal` procedure, in line 61 of Fig. 6. If the key-count is above this level, a new root node is allocated in line 63. The size of the children array is set to the square root of the key-count. Notably, the `degree` field is initially set to `0`, but it is later used in line 76 to enable threads to coordinate between the child slots that they work on, and is set to the proper value by the time the rebuilding completes.

Once the root node of the new subtree is allocated, threads compete to write it into the `newTarget` field of the `Rebuild` object, in line 69. After the new root is announced, threads atomically increment the `degree` field to select a child index to work on, in lines 73-78. Upon acquiring an index, a thread calls the `rebuildAndSetChild` procedure. This procedure calculates the interval of keys from the original subtree for the new child, and then calls `createIdeal` to create the child tree. The `createIdeal` procedure is not shown due to space reasons, but it is a straightforward traversal of the original tree – since the original is effectively immutable, no synchronization is necessary. After the new child is created, the thread runs a `DCSS` in line 98 to write the child into the array. If `DCSS` fails due to a change in the `status` field, then this means that another rebuild operation is ocurring higher in the tree. In this case, `rebuildAndSetChild` returns `false` to the caller, allowing it to stop rebuilding early.

```
59  procedure createIdealCollaborative(op, keyCount)
60    if keyCount < COLLABORATION_THRESHOLD then
61      newTarget = createIdeal(op.target, keyCount)
62    else
63      newTarget = new Inner(
64        initSize = keyCount,
65        degree = 0, // Will be set to final value in line 76.
66        keys = new KeyType[$\lfloor\sqrt{keyCount}\rfloor$ - 1],
67        children = new Node[$\lfloor\sqrt{keyCount}\rfloor$],
68        status = [0, ⊥, ⊥], count = 0, nextMark = 0)
69    if not CAS(op.newTarget, null, newTarget) then
70      // Subtree root was inserted by another thread.
71      newTarget = READ(op.newTarget)
72    if keyCount < COLLABORATION_THRESHOLD then
73      while true
74        index = READ(newTarget.degree)
75        if index == length(newTarget.children) then break
76        if CAS(newTarget.degree, index, index + 1) then
77          if not rebuildAndSetChild(op, keyCount, index)
78            return newTarget
79      for index in 0 until length(newTarget.children)
80        child = READ(newTarget.children[index])
81        if child == null then
82          if not rebuildAndSetChild(op, keyCount, index)
83            return newTarget
84    return newTarget
85
86  procedure rebuildAndSetChild(op, keyCount, index)
87    // Calculate the key interval for this child, and rebuild.
88    totalChildren = $\lfloor\sqrt{keyCount}\rfloor$
89    childSize = $\lfloor$keyCount / totalChildren$\rfloor$
90    remainder = keyCount % totalChildren
91    fromKey = childSize * index + min(index, remainder)
92    childKeyCount = childSize + (index < remainder ? 1 : 0)
93    child = createIdeal(op.target, fromKey, childKeyCount)
94    if index < length(op.newTarget.keys)
95      key = findKeyAtIndex(op.target, fromKey)
96      WRITE(op.newTarget.keys[index], key)
97    // Set new child, check if failed due to status change.
98    result = DCSS(op.newTarget.children[index],
99      null, child, op.newTarget.status, [0, ⊥, ⊥])
100   return result != FAILED_AUX_ADDRESS
```

Figure 6. The `createIdealCollaborative` Procedure

Notably, the key is written non-conditionally into the `keys` array in line 96, since potential helpers write the same value.

When the `degree` gets equal to the length of the `children` array, it means that some thread had started creating a new child at every entry of the array (moreover, some threads could have already created a new child, and set an entry in the `children` array to point to that new child). To guarantee lock-freedom, if a thread cannot increment `degree` further, then it must help the slow threads complete their own children. In lines 79-83, a thread checks the entries of the `children` array, and helps rebuild the children at entries whose value is still `null`. The rebuilding is completed once all the entries are non-`null`.

Collaborative building example. The collaborative subtree rebuilding is illustrated in Fig. 7D-G. After the `target` subtree is marked, and is determined to have 9 keys in total, the threads compete to announce the root of the new subtree with a `DCSS` instruction in Fig. 7D. The newly announced

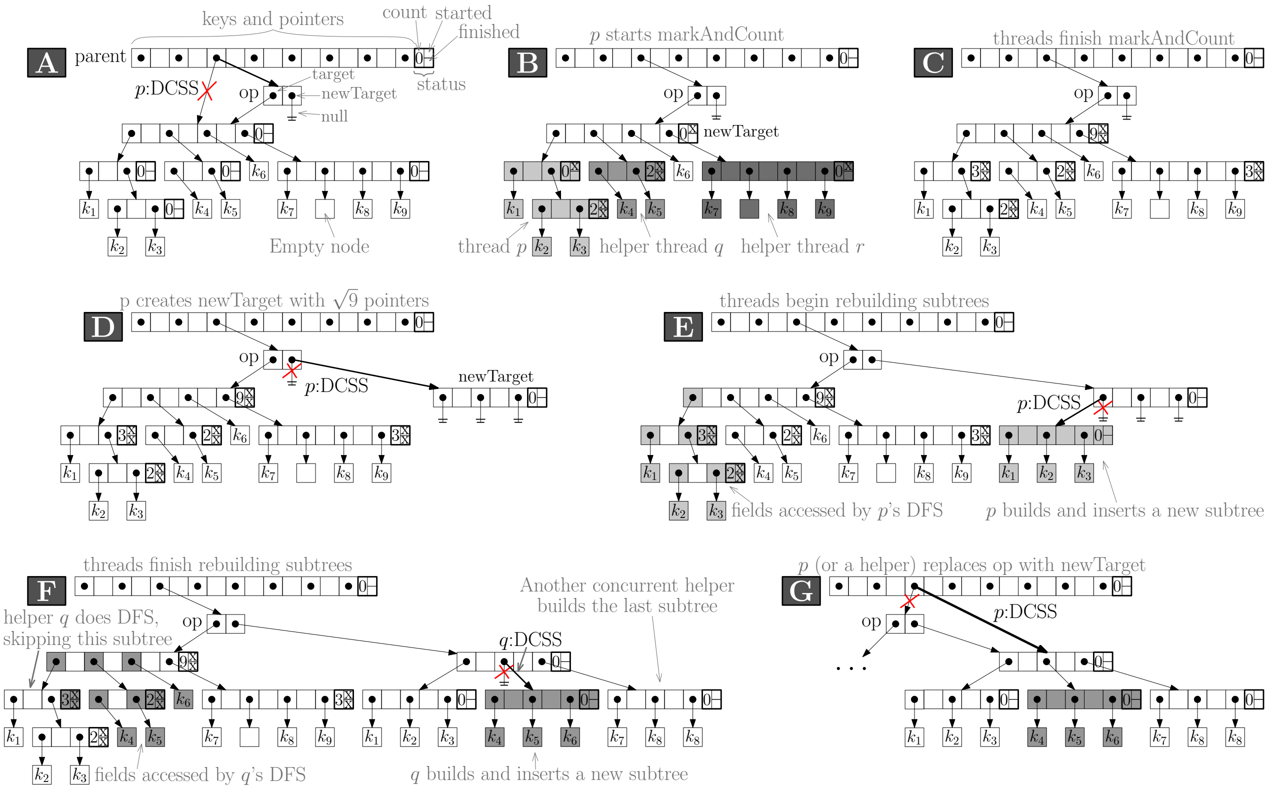

A
parent
keys and pointers
count started
finished
status
p:DCSS
op
target
newTarget
null
k_1
k_2
k_3
k_4
k_5
k_6
k_7
k_8
k_9
Empty node
B
p starts markAndCount
op
newTarget
thread p
helper thread q
helper thread r
C
threads finish markAndCount
op
D
p creates newTarget with $\sqrt{9}$ pointers
op
p:DCSS
newTarget
E
threads begin rebuilding subtrees
op
p:DCSS
fields accessed by p's DFS
p builds and inserts a new subtree
F
threads finish rebuilding subtrees
helper q does DFS, skipping this subtree
op
q:DCSS
Another concurrent helper builds the last subtree
fields accessed by q's DFS
q builds and inserts a new subtree
G
p (or a helper) replaces op with newTarget
op
p:DCSS
. . .

Figure 7. Illustration of the collaborative rebuilding algorithm.

node has $\sqrt{9} = 3$ entries, so each of its children will cover $9/\sqrt{9} = 3$ keys. In Fig. 7E, thread p acquired the index 0 of the `children` array, and determined that it needs to collect the keys k_1, k_2 and k_3 of the original subtree (shown in gray). Thread p allocated a new child node of size 3, and used `DCSS` to enter that child into the `children` array. In Fig. 7F, another thread had built and stored the child at index 2 of the `children` array, while the thread q is the only thread that is still working on the index 1. Helping threads can now enter the lines 79-83 of the pseudocode in Fig. 6, and can compete with the thread q to populate the index 1. In Fig. 7G, the rebuilding is completed, and the threads compete to replace the `Rebuild` object with the new, ideal subtree.

3.2 Lookups and Range Queries

The `lookup` subroutine, shown in Figure 8, is similar to the `insert`. An interpolation search is repeated until reaching an `Empty` or a `Single` node. If it reaches a `Single` node that contains the specified key, it returns `true`. Otherwise, if `lookup` encounters an `Empty` node or a `Single` node that does *not* contain the specified key, it returns `false`.

If `lookup` encounters a `Rebuild` object, it simply follows the `target` pointer to move to the next node, and continues traversal. Unlike the `insert` operation, `lookup` does not help concurrent subtree rebuilding operations. Lookups do not need to help rebuilding to ensure progress, and so they avoid the unnecessary overhead. Apart from its use of DCSS_READ and the handling of `Rebuild` objects, *lookup* is effectively a sequential interpolation tree search.

```
procedure lookup(ist, key)
  n = ist.root
  while true
    if n is Inner then
      index = interpolationSearch(key, n)
      n = DCSS_READ(n.children[index])
    else if n is Single then return n.k == key ? n.v : null
    else if n is Empty then return null
    else if n is Rebuild then n = n.target
```

Figure 8. Lookup Operation

Range queries. In some applications it is useful to have access to non-blocking *range query* operations, which return all of the keys in the data structure that intersect some range [*low*, *high*]. The IST could be augmented with support for range query operations using, for example, the recently introduced methodology of Arbel-Raviv and Brown [5].

4 Analysis

This section contains an outline of the correctness proofs and the complexity analysis of the C-IST data structure. For reasons of space, we only list the most important lemmas and

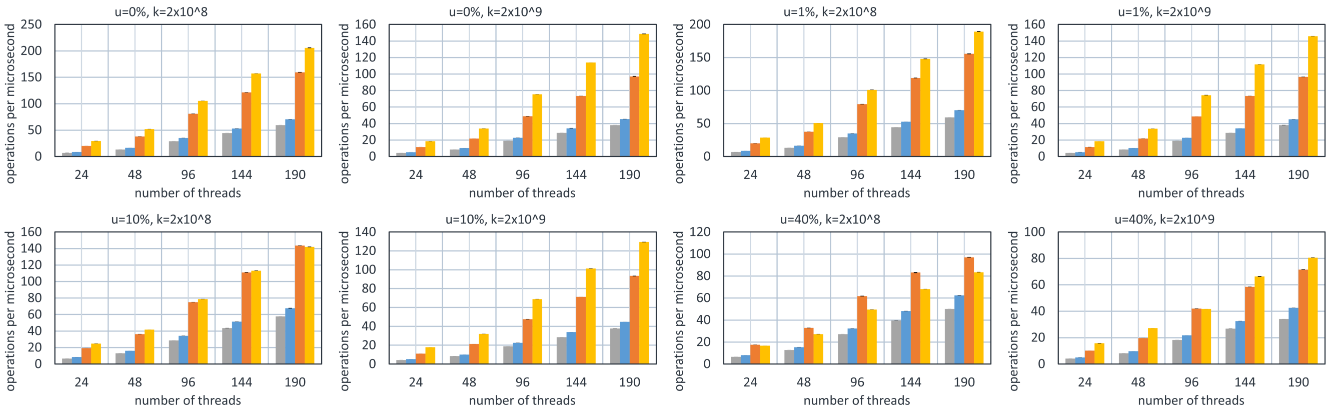


Figure 9. Basic Operations Throughput, Higher is Better (■ NM, 🟦 BCCO, 🟧 ABTree, 🟨 C-IST)

theorems. The full arguments are given in the full version of this paper [17].

4.1 Safety, Linearizability and Lock-Freedom

To prove correctness, we associate the C-IST and its operations with the semantics of an abstract set $\mathbb{A}$.

Definition 4.1 (Consistency)**.** An C-IST $\mathtt{I}$ is *consistent* with an abstract set $\mathbb{A}$ if and only if $\forall k \in \mathbb{A} \Leftrightarrow \text{hasKey}(\mathtt{I}, k)$.

By identifying the atomic instructions at which the corresponding abstract set $\mathbb{A}$ changes, we show that a C-IST operation changes the corresponding set exactly once. At the same time, we identify instructions that change the state of the data structure, but not the state of the corresponding abstract set. The linearizability proof follows naturally.

Theorem 4.2 (Safety)**.** *An C-IST $\mathtt{I}$ is always valid and consistent with some abstract set $\mathbb{A}$. C-IST operations are consistent with the operational semantics of the abstract sets.*

Corollary 4.3 (Linearizability)**.** *Lookup, insertion and deletion operations are linearizable.*

To show lock-freedom of the modification operations, we show that, for any C-IST, only finitely many data structure changes occur before the corresponding abstract set changes.

Lemma 4.4. *There is a finite number of steps between any two C-IST modifications, and there are finitely many consecutive modifications that do not change the abstract set.*

Theorem 4.5 (Non-Blocking)**.** *Insertion and deletion are lock-free, and lookup is wait-free.*

4.2 Complexity

The complexity analysis for C-IST follows the argument for sequential ISTs [34], with modifications due to the fact that at any time there can be up to p threads that are concurrently modifying the C-IST. The complete arguments can be found in the additional material. In particular, the worst-case depth bound is the following:

Lemma 4.6. *Let p be the number of concurrent threads that are modifying a C-IST. Worst-case depth of a C-IST that contains n keys is $O(p + \log n)$.*

In turn, a standard amortization argument implies the following naive worst-case amortized bound:

Lemma 4.7. *The worst-case amortized cost of insert and delete operations, without including the cost of searching for the node in the C-IST, is $O(\gamma(p + \log n))$, where γ is a bound on the average interval contention.*

The above worst-case bound can probably be further tightened. However, our main focus is on *expected amortized* bounds, which allow us to go below $\Theta(\log n)$. The following holds for the expected amortized cost of updates:

Lemma 4.8. *Let μ be a probability density with a finite support $[a, b]$. The expected total cost of processing a sequence of n μ-random insertions and uniformly random deletions into an initially empty C-IST is $O(n(\log \log n + p)\gamma)$, where γ is a bound on average interval contention.*

We note that, for worst-case schedules, the value of γ can be $\Theta(p)$, although in practice we expect it to be lower. For searches, the following holds:

Lemma 4.9. *Let μ be a smooth probability density, as defined Mehlkorn and Tsakalidis [34], for a parameter α, such that $\frac{1}{2} \leq \alpha < 1$. The expected search time in a μ-random IST of size n is $O(\log \log n + p)$.*

5 Evaluation

We implemented the concurrent IST in C++, and compared it against several state of the art concurrent data structures. We ran the benchmarks on a NUMA system with four Intel Xeon Platinum 8160 3.7GHz CPUs, each of which has 24 cores and 48 hardware threads. Within each CPU, cores share a 33MB LLC, and cores on different CPUs do not share any caches. The system has 384GB of RAM, and runs Ubuntu Linux 18.04.1 LTS. Our code was compiled with GCC 7.4.0-1, with

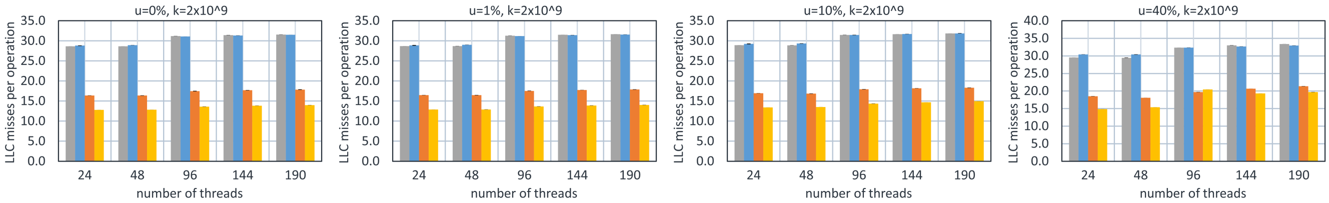


Figure 10. Last-Level Cache-Misses, Lower is Better (■ NM, ■ BCCO, ■ ABTree, ■ C-IST)

the highest optimization level (`-O3`). Threads were *pinned* to cores such that thread counts up to 48 ran on only one CPU, thread counts up to 96 run on only two CPUs, and so on. We used the fast scalable allocator jemalloc 5.0.1-25. When a memory page is allocated on our 4-CPU Xeon system, it has an *affinity* for a single CPU, and other CPUs pay a penalty to access it. We used the `numactl -interleave=all` option to ensure that pages are evenly distributed across CPUs.

We compared our IST implementation (C-IST) to the leading non-blocking binary search tree (NM) due to Natarajan and Mittal [36], Bronson's concurrent AVL tree [10] (BCCO), which is the leading blocking binary search tree, and a fast non-blocking (a, b)-tree (ABTree) due to Brown (Ch.8 of [13]), which is a concurrency-friendly variant of a B-tree. (*We also compared with many other concurrent search trees*, which are omitted here. See Section 5 in the corresponding technical report [52] for details.)

The goal of the evaluation section is to examine whether the amortized $O(\log \log n)$ running time induces performance improvements on datasets that are reasonably large. We therefore evaluate the C-IST operations against other comparable data structures in Section 5.1, where we show, for 1 billion keys, improvements ranging from 15-50% compared to the (a, b)-tree [13] (the next best alternative), depending on the ratio of updates and lookups. To further characterize the performance, we compare the average key depth and the impact on cache behavior in Section 5.2, and we show a breakdown of the execution time in Section 5.3. We conclude with a comparison of memory footprints in Section 5.4.

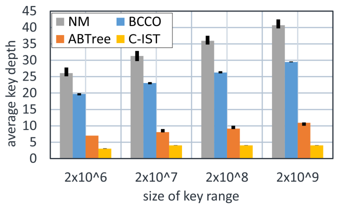
5.1 Comparison of the Basic Operations

Figure 9 shows the throughput of concurrent IST operations, compared against other sorted set data structures, for dataset sizes of $k = 2 \cdot 10^8$ and $k = 2 \cdot 10^9$ keys, and for $u = 0\%$, $u = 1\%$, $u = 10\%$ and $u = 40\%$, where u is the ratio of update operations among all operations. Plots for additional dataset sizes are shown in Figure 10 of the corresponding technical report [52].

In all cases, C-IST operations have much higher throughput than Natarajan and Mittal's non-blocking binary search tree (NM), and concurrent AVL trees due to Bronson (BCCO). For update ratios $u = 0\%$ and $u = 1\%$, concurrent IST also has a higher throughput compared to Brown's non-blocking (a, b)-tree. The underlying cause for better throughput is a lower rate of LLC misses due to IST's doubly-logarithmic depth. For higher update ratios $u = 10\%$ and $u = 40\%$, the cost of concurrent rebuilds starts to dominate the gains of doubly-logarithmic searches, and ABTree has a better throughput for $k = 2 \cdot 10^8$ keys. Above $k = 2 \cdot 10^9$ keys, C-IST outperforms ABTree even for the update ratio of $u = 40\%$.

5.2 Average Depth and Cache Behavior

The main benefit of C-IST's expected-$O(\log \log n)$ depth is that the key-search results in less cache misses compared to other tree data structures. The plot shown below compares the average number of pointer hops required to reach a key (error bars show min/max values over all trials), for dataset sizes from $2 \cdot 10^6$ to $2 \cdot 10^9$ keys. While the average depth is 20-40 for NM and BCCO, the average ABTree depth is between 6 and 10, and the average C-IST depth is below 5.



The differences in average depths between these data structures correlate with the average number of cache misses. Figure 10 compares the average number of last-level cache-misses between the different data structures, for different update ratios u. For the dataset size of $2 \cdot 10^9$ keys, C-IST operations undergo 2× less cache misses, and slightly fewer cache misses than ABTree. A detailed set of plots for different dataset sizes is shown in Figure 11 of the corresponding technical report [52].

5.3 Breakdown of the Execution Time

A breakdown of the execution time is shown in Figure 11, which contains plots for non-collaborative and collaborative rebuilding, update ratios $u = 10\%$ and $u = 40\%$, and the dataset size $2 \cdot 10^9$. In the non-collaborative variant, and for higher thread counts, the execution time is dominated by the useless helping operations. Since the work performed by the helping threads is discarded, this results in scalability issues as the update ratio u grows. In the collaborative variant, this problem does not occur, and most of the rebuilding time is spent in creating new subtrees. A more detailed set of plots is shown in Figure 12 and Figure 13 of the corresponding technical report [52].

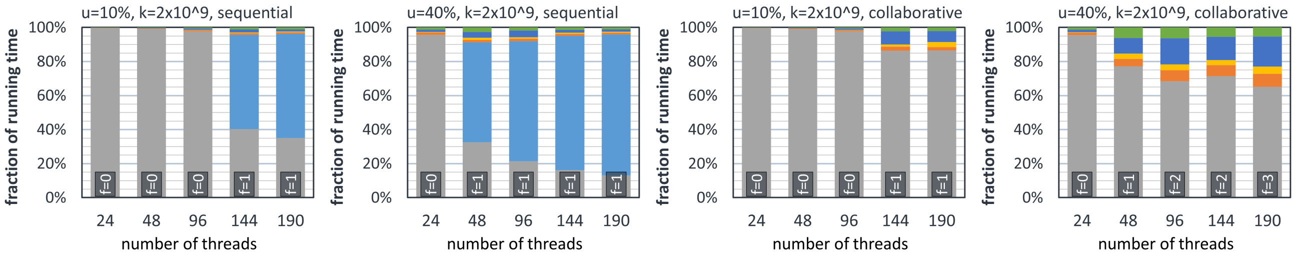


Figure 11. Execution Time Breakdown (🟦 creating, 🟩 marking, 🟦 useless helping, 🟧 deallocation, 🟨 locating garbage, ⬛ other). Bars are annotated with f, the number of times the *root* (entire tree) was rebuilt.

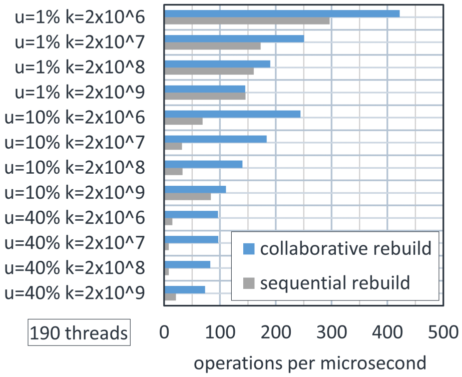


Figure 12. Comparison of Rebuilding Implementations

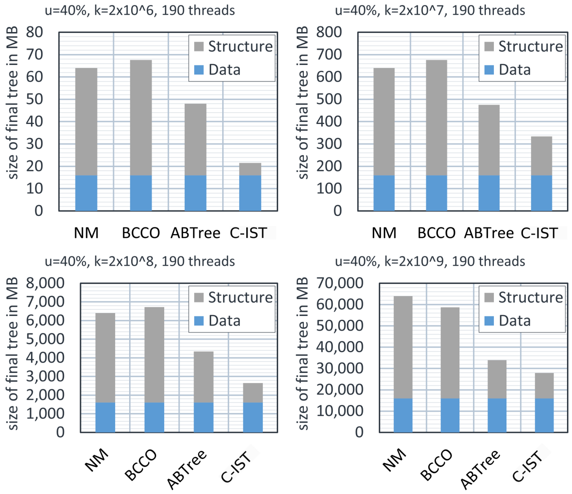


Figure 13. Memory Footprint Comparison

5.4 Memory Footprint

Due to using a lower number of nodes for the same dataset, the average space overhead is lower for the C-IST than the other data structures. Figure 13 shows the different memory footprints for four different dataset sizes. C-IST has a relative space overhead of ≈30-100%, whereas the overhead of the other data structures is between ≈120-400%.

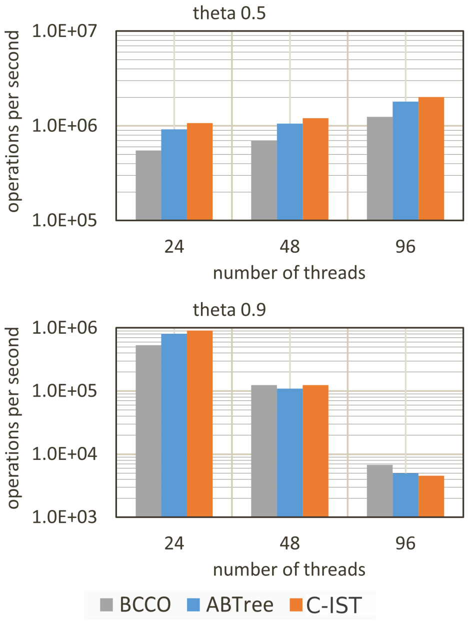


Figure 14. YCSB database performance with different index data structures, and a skewed *key access pattern*. (NM omitted because it is slower than BCCO.)

5.5 Additional Experiments

No-SQL database workload. We study a simple *in-memory database management system* called DBx1000 [63], which is used in multi-core database research. DBx implements a simple relational database, which contains one or more *tables*. Each table can have one or more *key fields* and associated *indexes*. Each index allows processes to query a specific key field, quickly locating any rows in which the key field contains a desired value. We replace the default index implementation in DBx with each of the BSTs that we study.

Following the approaches in [6, 63], we run a subset of the well known Yahoo! Cloud Serving Benchmark (YCSB) core with a single table containing 100 million rows, and a single index. Each thread performs a fixed number of transactions (100,000 in our runs), and the execution terminates when the first thread finishes performing its transactions. Each transaction accesses 16 different rows in the table, which are determined by index lookups on randomly generated keys. Each row is read with probability 0.9 and written with probability 0.1. The keys that a transaction will *access* are generated according to a **Zipfian** distribution following the approach in [27].

The results in Figure 14 show how performance degrades as the distribution of *accesses* to keys becomes highly skewed. (Higher θ values imply a more extreme skew. A θ value of 0.9 is *extremely* skewed.)

Trees containing Zipfian-distributed keys. Since the performance of the C-IST can theoretically degrade when the tree contains a highly skewed set of keys, we construct a synthetic benchmark to study such scenarios. In this benchmark, n threads access a single instance of the C-IST, and there is a *prefilling* phase followed by a *measured* phase. In the prefilling phase, each thread repeatedly generates a key from a Zipfian distribution ($\theta = 0.5$) over the key range $[1, 10^8]$ (picking one of 100 million *possible* keys), and inserts this key into the data structure (if it is not already present). This continues until the data structure contains 10 million keys (only 10% of the key range), at which point the prefilling phase ends. In the measured phase, all threads perform u% updates and $(100 - u)$% searches (for $u \in \{0, 1, 10\}$) on keys drawn from the same Zipfian distribution, for 30 seconds. This entire process is repeated for multiple trials, and for thread counts $n \in \{24, 48, 96, 144, 190\}$ (with at least one core left idle to run system processes). The results in Figure 15 suggest that the C-IST can remain robust even in scenarios where it contains a highly skewed distribution.

Artifact Evaluation. All code is publicly available, and a working artifact is submitted as part of this work.

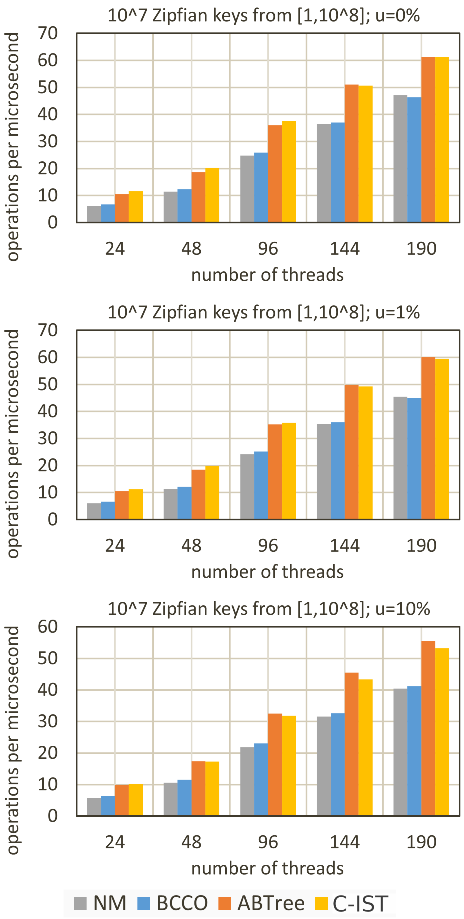


Figure 15. Synthetic benchmark in which the *set of keys stored* in a data structure is highly skewed.

6 Related Work

Sequential interpolation search was first proposed by Peterson [40], and subsequently analyzed by [25, 39, 62]. The *dynamic* case, where insertions and deletions are possible, was proposed by Frederickson [24]. The sequential IST variant we build on is by Mehlhorn and Tsakalidis [34]. This data structure supports amortized insertions and deletions in $O(\log n)$ time, under arbitrary distributions, and amortized insertion, deletion, and search, in $O(\log \log n)$ time under smoothness assumptions on the key distribution. To improve scalability, we augmented C-IST with parallel marking (to prevent updates during rebuilding), and a parallel rebuilding phase.

For *concurrent* search data structures ensuring predecessor queries, the work that is closest to ours is the SkipTrie [37], which allows predecessor queries in amortized expected $O(\log \log u + \gamma)$ steps, and insertions and deletions in $O(\gamma \log \log u)$ time, where u is the size of the key space, and γ is an upper bound on contention. The C-IST provides inferior runtime bounds in the worst case (e.g., $O(\log n)$ versus $O(\log \log u)$ amortized); however, the guarantees provided under distributional assumptions are asymptotically the same. We believe the C-IST should provide superior practical performance due to better cache behavior. We have attempted to provide a comparison of the C-IST with an open-source implementation of the SkipTrie [1]; we found that this implementation had significant stability and performance issues, which render a fair comparison impossible.

There is considerable work on designing efficient concurrent search tree data structures with predecessor queries, e.g. [6, 9, 11, 13, 14, 14, 22, 36]. The average-case complexity of these operations is usually logarithmic in the number of keys. For large key counts (our target application) this search term dominates, giving the C-IST a significant performance advantage. This effect is apparent in our experimental section.

Other work on concurrent search tree data structures includes early work by Kung [31], Bronson's lock-based concurrent AVL trees [10], Pugh's concurrent skip list [58], and later improvements by Herlihy et al. [29] (which the JDK implementation is based on), non-blocking BSTs due to Ellen et al. [23], and the KiWi data structure due to Basin et al. [8].

The DCSS and DCSS_READ primitives that we rely on were originally proposed by Harris [28]. The DCSS primitive needs to allocate a descriptor object to synchronize multiple memory locations. Our C++ implementation of DCSS, due to Arbel-Raviv and Brown [3], is able to recycle the descriptors. There are alternative primitives to DCSS with similar expressive power, such as the GCAS instruction [51], used to achieve snapshots in the Ctrie data structure.

Many concurrent data structures use the technique of snapshotting the entire data structure or some part thereof, with the goal of implementing a specific operation. The SnapQueue data structure [44] uses a *freezing* technique in which the writable locations are overwritten with special values such that the subsequent CAS operations fail. Ctries [43, 48, 49, 51] use the afore-mentioned GCAS operation to prevent further updates to the data structure. Work-stealing iterators [56, 57], used in work-stealing schedulers [33, 55] for data-parallel collections [50], use similar techniques to capture a snapshot of the iterator state.

The core motivation behind C-IST is to decrease the number of pointer hops during the key search. The underlying reason for this is that cache misses, which are incurred during the key search, are the dominating factor in the operation's running time. A recent trend in concurrent data structure design is to make data structures more *flat*, and in this way reduce the effect of the bottlenecks in the memory hierarchy. This is evident in that many data structures batch nodes within a single memory object – examples include concurrent search-tree designs [9, 15], lists, queues and ring buffers [41, 44, 61], unrolled skip lists [42], and tries [48]. A more recent flattening technique used in Cache-Tries is to include an auxiliary, quiescently-consistent table to speed up the key searches [45–47].

Our implementation of the C-IST data structure uses a scalable concurrent counter in the root node to track the number of updates since the last rebuild of the root node. In the past, a large body of research focused on scalable concurrent counters, both deterministic and probabilistic variant thereof [2, 7, 20, 21, 30, 60]. Scalable counters are useful in a number of other non-blocking data structures, which use counters to track their size or various statistics about the data structure. These include non-blocking queues [35], FlowPools [53, 54, 59], concurrent hash maps in the JDK [32], certain concurrent skip list implementations [26], and graphs with reachability queries [19].

Our C-IST implementation is done in C++, and it uses a custom concurrent memory management scheme due to Brown [12, 18]. In addition, our implementation uses techniques that decrease memory-allocator pressure by reusing the descriptors that are typically used in lock-free algorithms [3, 4].

7 Conclusion

We presented C-IST, the first concurrent implementation of a dynamic interpolation search tree. C-IST is non-blocking and scalable, and it preserves the desirable complexity properties of the original data structure with high probability. Experimental results in C++ suggest that C-IST significantly improves upon the performance of classic search data structures with similar semantics, by up to ≈ 3.5×, and the current best-performing alternative by up to 50%.

These findings suggest that concurrent data structure designs can be improved in non-trivial ways by exploiting input-specific techniques developed in the sequential case. We see this as an interesting line of potential future work.

Acknowledgments. This project has received funding from the European Research Council (ERC) under the European Union Horizon 2020 research and innovation program, grant agreement No 805223, ERC Starting Grant ScaleML.

We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC).

References

[1] 2018. SkipTrie Implementation at GitHub. https://github.com/JoeLeavitt/SkipTrie

[2] Dan Alistarh, Trevor Brown, Justin Kopinsky, Jerry Z. Li, and Giorgi Nadiradze. 2018. Distributionally Linearizable Data Structures. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures (SPAA '18)*. ACM, New York, NY, USA, 133–142. https://doi.org/10.1145/3210377.3210411

[3] Maya Arbel-Raviv and Trevor Brown. 2017. Reuse, Don't Recycle: Transforming Lock-Free Algorithms That Throw Away Descriptors. In *31st International Symposium on Distributed Computing, DISC 2017, October 16-20, 2017, Vienna, Austria*. 4:1–4:16. https://doi.org/10.4230/LIPIcs.DISC.2017.4

[4] Maya Arbel-Raviv and Trevor Brown. 2017. Reuse, don't Recycle: Transforming Lock-free Algorithms that Throw Away Descriptors. *CoRR* abs/1708.01797 (2017). arXiv:1708.01797 http://arxiv.org/abs/1708.01797

[5] Maya Arbel-Raviv and Trevor Brown. 2018. Harnessing Epoch-based Reclamation for Efficient Range Queries. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '18)*. ACM, New York, NY, USA, 14–27. https://doi.org/10.1145/3178487.3178489

[6] Maya Arbel-Raviv, Trevor Brown, and Adam Morrison. 2018. Getting to the Root of Concurrent Binary Search Tree Performance. In *USENIX Annual Technical Conference*.

[7] James Aspnes, Maurice Herlihy, and Nir Shavit. 1994. Counting Networks. *J. ACM* 41, 5 (Sept. 1994), 1020–1048. https://doi.org/10.1145/185675.185815

[8] Dmitry Basin, Edward Bortnikov, Anastasia Braginsky, Guy Golan-Gueta, Eshcar Hillel, Idit Keidar, and Moshe Sulamy. 2017. KiWi: A Key-Value Map for Scalable Real-Time Analytics. In *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '17)*. ACM, New York, NY, USA, 357–369. https://doi.org/10.1145/3018743.3018761

[9] Anastasia Braginsky and Erez Petrank. 2012. A lock-free B+ tree. In *Proceedings of the twenty-fourth annual ACM symposium on Parallelism in algorithms and architectures*. ACM, 58–67.

[10] Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. 2010. A Practical Concurrent Binary Search Tree. *SIGPLAN Not.* 45, 5

(Jan. 2010), 257–268. https://doi.org/10.1145/1837853.1693488

[11] Trevor Brown. 2014. B-slack Trees: Space Efficient B-Trees. In *Algorithm Theory - SWAT 2014 - 14th Scandinavian Symposium and Workshops, Copenhagen, Denmark, July 2-4, 2014. Proceedings*. 122–133. https://doi.org/10.1007/978-3-319-08404-6_11

[12] Trevor Brown. 2017. Reclaiming memory for lock-free data structures: there has to be a better way. *CoRR* abs/1712.01044 (2017). arXiv:1712.01044 http://arxiv.org/abs/1712.01044

[13] Trevor Brown. 2017. *Techniques for Constructing Efficient Data Structures*. Ph.D. Dissertation. University of Toronto.

[14] Trevor Brown and Hillel Avni. 2012. Range Queries in Non-blocking *k*-ary Search Trees. In *Principles of Distributed Systems, 16th International Conference, OPODIS 2012, Rome, Italy, December 18-20, 2012. Proceedings*. 31–45. https://doi.org/10.1007/978-3-642-35476-2_3

[15] Trevor Brown and Joanna Helga. 2011. Non-blocking k-ary Search Trees. In *Principles of Distributed Systems - 15th International Conference, OPODIS 2011, Toulouse, France, December 13-16, 2011. Proceedings*. 207–221. https://doi.org/10.1007/978-3-642-25873-2_15

[16] Trevor Brown, Aleksandar Prokopec, and Dan Alistarh. 2020. Artifact for Non-Blocking Interpolation Search Trees with Doubly-Logarithmic Running Time. https://doi.org/10.1145/3332466.3374542

[17] Trevor Brown, Aleksandar Prokopec, and Dan Alistarh. 2020. Non-Blocking Interpolation Search Trees with Doubly-Logarithmic Running Time. In *Proceedings of the 25th Symposium on Principles and Practice of Parallel Programming (PPoPP '20)*. ACM, New York, NY, USA.

[18] Trevor Alexander Brown. 2015. Reclaiming Memory for Lock-Free Data Structures: There has to be a Better Way. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing, PODC 2015, Donostia-San Sebastián, Spain, July 21 - 23, 2015*. 261–270. https://doi.org/10.1145/2767386.2767436

[19] Bapi Chatterjee, Sathya Peri, Muktikanta Sa, and Nandini Singhal. 2019. A Simple and Practical Concurrent Non-Blocking Unbounded Graph with Linearizable Reachability Queries. In *Proceedings of the 20th International Conference on Distributed Computing and Networking (ICDCN '19)*. Association for Computing Machinery, New York, NY, USA, 168–177. https://doi.org/10.1145/3288599.3288617

[20] Damian Dechev and Bjarne Stroustrup. 2009. Scalable Nonblocking Concurrent Objects for Mission Critical Code. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications (OOPSLA '09)*. Association for Computing Machinery, New York, NY, USA, 597–612. https://doi.org/10.1145/1639950.1639954

[21] Dave Dice, Yossi Lev, and Mark Moir. 2013. Scalable Statistics Counters. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '13)*. Association for Computing Machinery, New York, NY, USA, 43–52. https://doi.org/10.1145/2486159.2486182

[22] Dana Drachsler, Martin Vechev, and Eran Yahav. 2014. Practical concurrent binary search trees via logical ordering. *ACM SIGPLAN Notices* 49, 8 (2014), 343–356.

[23] Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. 2010. Non-blocking Binary Search Trees. In *Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC '10)*. ACM, New York, NY, USA, 131–140. https://doi.org/10.1145/1835698.1835736

[24] Greg N Frederickson. 1983. Implicit data structures for the dictionary problem. *Journal of the ACM (JACM)* 30, 1 (1983), 80–94.

[25] Gaston H Gonnet, Lawrence D Rogers, and J Alan George. 1980. An algorithmic and complexity analysis of interpolation search. *Acta Informatica* 13, 1 (1980), 39–52.

[26] Vincent Gramoli. 2015. More than You Ever Wanted to Know about Synchronization: Synchrobench, Measuring the Impact of the Synchronization on Concurrent Algorithms. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2015)*. Association for Computing Machinery, New York, NY, USA, 1–10. https://doi.org/10.1145/2688500.2688501

[27] Jim Gray, Prakash Sundaresan, Susanne Englert, Ken Baclawski, and Peter J. Weinberger. 1994. Quickly Generating Billion-record Synthetic Databases. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data (SIGMOD '94)*. ACM, New York, NY, USA, 243–252. https://doi.org/10.1145/191839.191886

[28] Timothy L. Harris, Keir Fraser, and Ian A. Pratt. 2002. A Practical Multi-word Compare-and-Swap Operation. In *Proceedings of the 16th International Conference on Distributed Computing (DISC '02)*. Springer-Verlag, Berlin, Heidelberg, 265–279. http://dl.acm.org/citation.cfm?id=645959.676137

[29] Maurice Herlihy, Yossi Lev, Victor Luchangco, and Nir Shavit. 2006. A Provably Correct Scalable Concurrent Skip List.

[30] Maurice Herlihy, Beng-Hong Lim, and Nir Shavit. 1995. Scalable Concurrent Counting. *ACM Trans. Comput. Syst.* 13, 4 (Nov. 1995), 343–364. https://doi.org/10.1145/210223.210225

[31] H. T. Kung and Philip L. Lehman. 1980. Concurrent Manipulation of Binary Search Trees. *ACM Trans. Database Syst.* 5, 3 (Sept. 1980), 354–382. https://doi.org/10.1145/320613.320619

[32] Doug Lea. 2018. Doug Lea's Workstation. http://g.oswego.edu/

[33] Jonathan Lifflander, Sriram Krishnamoorthy, and Laxmikant V. Kale. 2013. Steal Tree: Low-Overhead Tracing of Work Stealing Schedulers. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 507–518. https://doi.org/10.1145/2491956.2462193

[34] Kurt Mehlhorn and Athanasios Tsakalidis. 1993. Dynamic interpolation search. *Journal of the ACM (JACM)* 40, 3 (1993), 621–634.

[35] Maged M. Michael and Michael L. Scott. 1996. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing (PODC '96)*. Association for Computing Machinery, New York, NY, USA, 267–275. https://doi.org/10.1145/248052.248106

[36] Aravind Natarajan and Neeraj Mittal. 2014. Fast Concurrent Lock-free Binary Search Trees. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '14)*. 317–328.

[37] Rotem Oshman and Nir Shavit. 2013. The SkipTrie: Low-depth Concurrent Search Without Rebalancing. In *Proceedings of the 2013 ACM Symposium on Principles of Distributed Computing (PODC '13)*. ACM, New York, NY, USA, 23–32. https://doi.org/10.1145/2484239.2484270

[38] Yehoshua Perl, Alon Itai, and Haim Avni. 1978. Interpolation Search – a Log logN Search. *Commun. ACM* 21, 7 (July 1978), 550–553. https://doi.org/10.1145/359545.359557

[39] Yehoshua Perl and Edward M Reingold. 1977. Understanding the complexity of interpolation search. *Inform. Process. Lett.* 6, 6 (1977), 219–222.

[40] W Wesley Peterson. 1957. Addressing for random-access storage. *IBM journal of Research and Development* 1, 2 (1957), 130–146.

[41] Kenneth Platz, Neeraj Mittal, and Subbarayan Venkatesan. 2014. Practical Concurrent Unrolled Linked Lists Using Lazy Synchronization. In *Principles of Distributed Systems*, Marcos K. Aguilera, Leonardo Querzoni, and Marc Shapiro (Eds.). Springer International Publishing, Cham, 388–403.

[42] Kenneth Platz, Neeraj Mittal, and S. Venkatesan. 2019. Concurrent Unrolled Skiplist. *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)* (2019), 1579–1589.

[43] Aleksandar Prokopec. 2014. Data Structures and Algorithms for Data-Parallel Computing in a Managed Runtime. (2014).

[44] Aleksandar Prokopec. 2015. SnapQueue: Lock-free Queue with Constant Time Snapshots. In *Proceedings of the 6th ACM SIGPLAN Symposium on Scala (SCALA 2015)*. ACM, New York, NY, USA, 1–12.

https://doi.org/10.1145/2774975.2774976

[45] Aleksandar Prokopec. 2017. Analysis of Concurrent Lock-Free Hash Tries with Constant-Time Operations. *ArXiv e-prints* (Dec. 2017). arXiv:cs.DS/1712.09636

[46] Aleksandar Prokopec. 2018. Cache-tries: Concurrent Lock-free Hash Tries with Constant-time Operations. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '18)*. ACM, New York, NY, USA, 137–151. https://doi.org/10.1145/3178487.3178498

[47] Aleksandar Prokopec. 2018. *Efficient Lock-Free Removing and Compaction for the Cache-Trie Data Structure.* Springer International Publishing, Cham.

[48] Aleksandar Prokopec, Phil Bagwell, and Martin Odersky. 2011. *Cache-Aware Lock-Free Concurrent Hash Tries.* Technical Report.

[49] Aleksandar Prokopec, Phil Bagwell, and Martin Odersky. 2011. *Lock-Free Resizeable Concurrent Tries.* Springer Berlin Heidelberg, Berlin, Heidelberg, 156–170. https://doi.org/10.1007/978-3-642-36036-7_11

[50] Aleksandar Prokopec, Phil Bagwell, Tiark Rompf, and Martin Odersky. 2011. A Generic Parallel Collection Framework. In *Proceedings of the 17th international conference on Parallel processing - Volume Part II (Euro-Par'11)*. Springer-Verlag, Berlin, Heidelberg, 136–147. http://dl.acm.org/citation.cfm?id=2033408.2033425

[51] Aleksandar Prokopec, Nathan Grasso Bronson, Phil Bagwell, and Martin Odersky. 2012. Concurrent Tries with Efficient Non-blocking Snapshots. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '12)*. ACM, New York, NY, USA, 151–160. https://doi.org/10.1145/2145816.2145836

[52] Aleksandar Prokopec, Trevor Brown, and Dan Alistarh. 2020. Analysis and Evaluation of Non-Blocking Interpolation Search Trees. (Dec. 2020). arXiv:cs.DS/2001.00413

[53] Aleksandar Prokopec, Heather Miller, Philipp Haller, Tobias Schlatter, and Martin Odersky. 2012. *FlowPools: A Lock-Free Deterministic Concurrent Dataflow Abstraction, Proofs.* Technical Report.

[54] Aleksandar Prokopec, Heather Miller, Tobias Schlatter, Philipp Haller, and Martin Odersky. 2012. FlowPools: A Lock-Free Deterministic Concurrent Dataflow Abstraction. In *LCPC*. 158–173.

[55] Aleksandar Prokopec and Martin Odersky. 2014. *Near Optimal Work-Stealing Tree Scheduler for Highly Irregular Data-Parallel Workloads.* Springer International Publishing, Cham, 55–86. https://doi.org/10.1007/978-3-319-09967-5_4

[56] Aleksandar Prokopec, Dmitry Petrashko, and Martin Odersky. 2014. On Lock-Free Work-stealing Iterators for Parallel Data Structures. (2014), 10. http://infoscience.epfl.ch/record/196627

[57] A. Prokopec, D. Petrashko, and M. Odersky. 2015. Efficient Lock-Free Work-Stealing Iterators for Data-Parallel Collections. In *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. 248–252. https://doi.org/10.1109/PDP.2015.65

[58] William Pugh. 1990. *Concurrent Maintenance of Skip Lists.* Technical Report. College Park, MD, USA.

[59] Tobias Schlatter, Aleksandar Prokopec, Heather Miller, Philipp Haller, and Martin Odersky. 2012. Multi-Lane FlowPools: A Detailed Look. (2012), 13.

[60] Guy L. Steele and Jean-Baptiste Tristan. 2016. Adding Approximate Counters. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '16)*. Association for Computing Machinery, New York, NY, USA, Article Article 15, 12 pages. https://doi.org/10.1145/2851141.2851147

[61] Martin Thompson, Dave Farley, Michael Barker, Patricia Gee, and Andrew Stewart. 2011. Disruptor: High performance alternative to bounded queues for exchanging data between concurrent threads. http://lmax-exchange.github.io/disruptor/files/Disruptor-1.0.pdf

[62] Andrew C Yao and F Frances Yao. 1976. The complexity of searching an ordered random table. In *Foundations of Computer Science, 1976., 17th Annual Symposium on.* IEEE, 173–177.

[63] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. 2014. Staring into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores. *VLDB* 8, 3 (Nov. 2014).

A Artifact Description

A.1 Getting Started Guide

We prepared a Docker container that can be used to run the experiments from the paper. The requirements are:

- A relatively recent Linux distribution (we used Ubuntu to prepare the artifact).
- An installation of a recent Docker (see instructions further below).
- A multicore CPU. We used a 4-socket system, with four Intel 8160 CPUs (192 threads total). A less parallel CPU will also suffice, although you will replicate less of our experiments.
- If you want to reproduce all the results, we suggest to have 384 GB of RAM memory, which is what our machine had available. If you have less RAM, for example 128 GB, the experiments might still work. Note that it is also possible to reduce the dataset sizes in most experiments (for details, see instructions in the Step-by-Step guide).

The artifact is available for download at Zenodo [16], at the URL https://zenodo.org/record/3600160#.XhReF2bQi5M.

The steps to download and run the artifact are as follows (note – you might have to use `sudo` for Docker):

1. Install a newer version of Docker to your system. For example, we were using ‘Docker version 19.03.5, build 633a0ea838‘. You can run:

   ```
   $ docker --version
   ```

   to check the version. If you are using the Ubuntu distribution, see the instructions here: https://docs.docker.com/install/linux/docker-ce/ubuntu/
2. Download the `ppopp20-artifact.tar.gz` file that is the docker image from: https://www.dropbox.com/s/xs34t8mmi53e6oh/paper-241.tar.gz?dl=0
3. Load the docker image from the downloaded file:

   ```
   $ docker load -i paper-241.tar.gz
   ```

4. Run the following to check that the image was loaded:

   ```
   $ docker images
   ```

   Start a Docker container from the artifact image. Note that you *have to run in the privileged mode* so that the artifact can use thread-to-CPU pinning and some other facilities. Run the following:

   ```
   $ docker run -i -t --privileged \
       ppopp20-artifact /bin/bash
   ```

5. Go to the `root/artifact` folder:

   ```
   $ cd /root/artifact
   ```

6. Run `ls`. You should see several folders, in particular `microbench` and `macrobench`. In this image, the source files have already been compiled for both the microbenchmarks and the macrobenchmarks, but if you want to compile them again, you should delete `microbench/bin` and `macrobench/bin`, then run `microbench/compile.sh`, and then to compile macrobenchmarks, run `macrobench/compile.sh`.
7. To test that you can run the benchmarks, please first go to `microbench/experiments`. Then run `ls`. You will see several folders, one for each experiment. Run:

   ```
   $ cd istree_exp1_scaling_threads
   ```

   to enter the first experiment. If necessary, make the `run.sh` script executable like this: `chmod a+x run.sh`. Then run the script:

   ```
   $ ./run.sh
   ```

   You should see output like this.

   ```
   Estimated 5 hours to run
   filename,DS_TYPENAME,size_node, ...
   step 10001/10128: ...
   step 10002/10128: ...
   ...
   ```

   Press CTRL-Z, run `top` and kill the `run.sh` process.
8. Now go to the `macrobench/experiments` folder, and the macro benchmark.

   ```
   $ cd /root/artifact/macrobench/\
     experiments/istree_exp1
   $ ./run.sh "< thread-counts >"
   ```

   where the `< thread-counts >` is the list of thread counts you want to run it with. For example, if your CPU has 8 cores, you can run it with `"2 4 8"`. Please make sure to include the double quotes. Also, do not use thread counts larger than the number of CPUs, because the `taskset` command, which is used in the benchmark, will fail. You should see output like this:

   ```
   Estimated running time ...

    alg  nthreads  theta ...
   ...
   ```

   Press CTRL-Z, run `top` and kill the `run.sh` process.

If you managed to run these steps, then you should be able to run the experiments.

A.2 Step-by-Step Instructions

The artifact supports the following claims from the paper:

- microbench/experiments/istree_exp1_scaling_threads – supports Figure 9, which shows the throughput of the basic operations for different data structures.
- microbench/experiments/istree_exp2_memory_static – supports Figure 13, which shows the memory footprints of the different data structures.
- istree_exp3_disable_multicounter in microbench/experiments/ – supports Figure 14 from the technical report [52], which evaluates the effect of multicounters.
- istree_exp4_disable_rebuild_helping in microbench/-experiments/ – supports Figure 12, which shows the effect of disabling collaborative rebuilding.
- microbench/experiments/istree_exp6_rebuilding_time – supports Figure 11, which shows the amount of time spent in C-IST rebuilding vs in other operations.
- microbench/experiments/istree_exp9_zipf – supports Figure 15, which shows the performance of basic ops on Zipfian key distributions.
- macrobench/experiments/istree_exp1 – supports Figure 14, which shows the performance of different data structures on the YCSB database benchmark.

Of these experiments, the most important is 'microbench/-experiments/istree_exp1_scaling_threads'.

To run each of these experiments:

1. Enter the respective folder.
2. Run the './run.sh' script (set permissions if necessary).
3. After the experiment completes, inspect the CSV files in that folder.

Note that, for each experiment, you can modify the 'run.sh' scripts to change various parameters of the experiment. For example, in 'istree_exp1_scaling_threads', you can change the 'thread_counts' variable to manually set the thread counts that you want to run with. Similarly, in the experiment 'istree_exp2_memory_static', you can change the value of 'key_range_sizes' to change the key counts.

When an experiment completes, it will produce a CSV file, which contains all the data. You can manually inspect the numbers in each CSV file, or you can open the Excel spreadsheet in each folder, paste the CSV file contents into it, and the graphs will automatically be generated for you if you click on the "refresh all" button (you need Excel macros to run for this, you can check them if you press ALT-F11).

The artifact technically does not reproduce the hardware counter performance, since we did not manage to run them within Docker. However, if you copy the folder with the artifact to your host system (`docker cp`), recompile everything there, and re-run the benchmarks, then you should be able to reproduce the hardware counter numbers (this data is included the CSV file of istree_exp1_scaling_threads microbenchmark).